# System Description: CyNTHIA

# System Description: $C^{Y}NTHIA$

Jon Whittle[1]*, Alan Bundy[1], Richard Boulton[1], and Helen Lowe[2]**

[1] Division of Informatics, University of Edinburgh, 80 South Bridge, Edinburgh EH1 1HN, Scotland.
[2] Dept. of Computer Studies, Glasgow Caledonian University, City Campus, Cowcaddens Road, Glasgow G4 0BA, Scotland.
jonathw@ptolemy.arc.nasa.gov

Current programming environments for novice functional programming (FP) are inadequate. This paper describes ways of using proofs as a foundation to improve the situation, in the context of the language ML [4]. The most common way to write ML programs is via a text editor and compiler (such as the Standard ML of New Jersey compiler). But program errors, in particular type errors, are generally difficult to track down. For novices, the lack of debugging support forms a barrier to learning FP concepts [5].

$C^{Y}NTHIA$ is an editor for a subset of ML that provides improved support for novices. Programs are created incrementally using a collection of correctness-preserving editing commands. Users start with an existing program which is adapted by using the commands. This means fewer errors are made. $C^{Y}NTHIA$'s improved error-feedback facilities enable errors to be corrected more quickly. Specifically, $C^{Y}NTHIA$ provides the following correctness guarantees:

1. syntactic correctness;
2. static semantic correctness, including type correctness as well as checking for undeclared variables or functions, or duplicate variables in patterns etc.;
3. well-definedness — all patterns are exhaustive and have no redundant matches;
4. termination.

Note that, in contrast to the usual approach, correctness-checking is done incrementally. Violations of (1), (3) and (4) can never be introduced into $C^{Y}NTHIA$ programs. (2) may be violated as in general it is impossible to transform one program into another without passing through states containing such errors. However, all static semantic errors are highlighted to the user by colouring expressions in the program text. The incremental nature of $C^{Y}NTHIA$ means that as soon as an error is introduced, it is indicated to the user, although the user need not change it immediately.

In $C^{Y}NTHIA$, each ML function definition is represented as a proof of a specification of that function, using the idea of proofs-as-programs [2]. As editing commands are applied, the proof is developed hand-in-hand with the program, as given in Fig. 1. The user starts with an existing program and a corresponding

* This author is now with Recom Technologies, NASA Ames Research Center, Moffet Field. CA 94035

** First author supported by EPSRC studentship and 519-50-32 NASA Code R Aero IT Base Program, SIPS, Program Synthesis. Other authors EPSRC grant GL/L/11724

initial proof (from an initial library). The edits are actually applied to the proof, giving a new partial proof which may contain gaps or inconsistencies. $C^{Y}NTHIA$ attempts to fill these gaps and resolve inconsistencies. Any which cannot be resolved are fed back to the user as program errors.

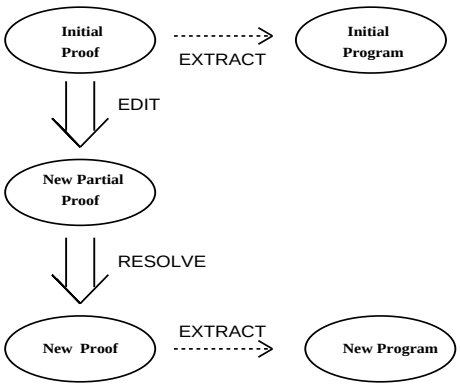


**Fig. 1.** Editing Programs in $C^{Y}NTHIA$.

$C^{Y}NTHIA$'s proofs are written in *Oyster* [1], a proof-checker implementing a variant of Martin-Löf Type Theory. *Oyster* specifications (or conjectures) may be written to any level of detail, but to make the proof process tractable in real-time, $C^{Y}NTHIA$ specifications are restricted severely. Specifications state precisely the type of the function and various lemmas needed for termination analysis. Proofs of such specifications provide guarantees (1)-(4) above. Given this restriction, all theorem proving can be done automatically.

We only consider a functional subset of the Core ML language [5]. In addition, we exclude mutual recursion and type inference. Mutual recursion could be added by extending the termination checker. We made a conscious decision to insist that the user provide type declarations. This is because the system is primarily intended for novices and investigations have shown that students find type inference confusing [5].

## 1 An Example of Using $C^{Y}NTHIA$

To illustrate the idea, consider the task of writing a function, `count`, to count the number of nodes in a binary tree, where the definition of the datatype `tree` is given in ML as:

```
datatype tree = leaf of int | node of int * tree * tree;
```

Suppose the user recognises that a function, `length`, to count the number of items in an integer list, is similar to the desired function. He[1] can then use `length` as a starting point. Below we give the definition of `length` preceded by its type

[1] We refer to the user by the pronoun 'he' although the user may be male or female.

```
'a list -> int
fun length nil = 0
|   length (x::xs) = 1 + (length xs);
```

Note that `'a list` is the polymorphic list type. We show how `length` could be edited into `count`. The user may indicate any occurrence of `length` and invoke the RENAME command to change `length` to `count`. $C^{Y}NTHIA$ then changes all other occurrences of `length` to `count`:

```
'a list -> int
 fun count nil = 0
 |   count (x::xs) = 1 + (count xs);
```

We want to count nodes in a tree so we need to change the type of the parameter. Suppose the user selects `nil` and invokes CHANGE TYPE to change the type to `tree`. $C^{Y}NTHIA$ propagates this change by changing `nil` to `leaf n` and changing `::` to `node`:

```
tree -> int
fun count (leaf n) = 0
|   count (node(x,xs,ys)) = 1 + (count xs);
```

A new variable `ys` of type `tree` has been introduced. In addition, `count ys` is made available for use as a recursive call in the program. It remains to alter the results for each pattern. `0` is easily changed to `1` using CHANGE TERM. If the user then clicks on `1` in the second line, a list of terms appear which include `count ys`. Selecting this term produces the final program:

```
tree -> int
fun count (leaf n) = 1
|   count (node(x,xs,ys)) = 1 + (count ys) + (count xs);
```

## 2 Representing ML Definitions as Proofs

Each ML function is represented by a proof with specification (i.e. top-level goal) that is precisely the type of the function along with lemmas required for termination analysis. In general, such specifications may specify arbitrarily complex behaviour about the function. However, $C^{Y}NTHIA$ specifications are deliberately rather weak so that the theorem proving task can be automated. The definition of quicksort given in Fig. 2 is easily represented in $C^{Y}NTHIA$.

A specification for `partition` would be as follows:

$$\exists P : (\forall z_1 : (int * int \ \rightarrow bool).\ \forall z_2 : int.\ \forall z_3 : int\ list. \\ (f\ z_1\ z_2\ z_3) : int\ list\ \wedge\ (f\ z_1\ z_2\ z_3) \leq_w z_3) \qquad (1)$$

where $f$ represents the name of the function and $P$ is a variable representing the definition of the ML function. $P$ gets instantiated as the inference rules are applied. A complete proof instantiates $P$ to a complete program. This is a standard approach to extracting programs from proofs. The first part of the

```
(int * int -> bool) -> int -> int list -> int list
fun partition f k nil = nil
|  partition f k (h::t) = if f(h,k) then h::partition f k t
                          else partition f k t;


int list -> int list
fun qsort nil = nil
|  qsort (h::t) = (qsort (partition (op <) h t)) @ [h]
                        @ (qsort (partition (op >=) h t));
```

**Fig. 2.** A Version of Quicksort.

(weak) specification merely states the existence of a function of the given type. The last conjunct specifies a termination condition.

$CYNTHIA$'s termination analysis is an extension of Walther Recursion [3] to ML. Walther Recursive functions form a decidable subset of the set of terminating functions, including primitive recursive functions over an inductively-defined datatype, nested recursive functions and some functions with previously defined functions in a recursive call, such as `qsort`. Walther Recursion assumes a fixed size ordering: $w(c(u_1, \ldots, u_n)) = 1 + \sum_{i \in R_c} w(u_i)$ where $c$ is a constructor and $R_c$ is the set of recursive arguments of $c$.

There are two parts to Walther Recursion — reducer / conserver (RC) analysis and measure argument (MA) analysis. Every time a new definition is made, RC lemmas are calculated for the definition. These place a bound on the definition based on the fixed size ordering. In (1), a conserver lemma for `partition` is $(f\ z_1\ z_2\ z_3) \leq_w z_3$ which says that the result of `partition` has size no greater than its third argument. To guarantee termination, it is necessary to consider each recursive call of a definition and show that the recursive arguments decrease with respect to this ordering. Since recursive arguments may in general involve references to other functions, showing a measure decrease may refer to previously derived RC lemmas. In `qsort`, for example, we need, amongst other things, to show that `partition (op >=) h t` $\leq_w$ `t`. This is achieved by using the lemma for `partition`.

Walther Recursion is particularly appropriate because $CYNTHIA$ is meant for novices who have no knowledge of theorem proving. The set of Walther Recursive functions is decidable. Hence, termination analysis is completely automated.

Clearly, there are an infinite number of proofs of a specification such as (1). The particular function represented in the proof is given by the user, however, since each editing command application corresponds to the application of a corresponding inference rule. In addition, many possible proofs are outlawed because the proof rules (and corresponding editing commands) have been designed in such a way as to restrict to certain kinds of proofs, namely those that correspond to ML definitions.

Formula (1) can be proved in a backwards fashion. The main ingredients of this proof are induction, to represent the recursion in `partition`, along with var-

ious correctness-checking rules which perform type-checking, termination analysis etc. The structure of the program is mirrored in the proof because the user drives the proof indirectly by applying editing commands to the program.

The use of proofs to represent ML programs is a flexible framework within which to carry out various kinds of analyses of the programs. It allows changes at multiple places in the program to be achieved by a single change to the proof, e.g. the induction scheme captures the recursion pattern of the function.

## 3 Evaluating *CYNTHIA*

*CYNTHIA* has been successfully evaluated in two trials at Napier University [5]. Although some semi-formal experiments were undertaken, most analysis was done informally. However, the following trends were noted:

- Students make fewer errors when using *CYNTHIA* than when using a traditional text editor.
- When errors are made, users of *CYNTHIA* locate and correct the errors more quickly. This especially applies to type errors.
- *CYNTHIA* discourages aimless hacking. The restrictions imposed by the editing commands mean that students are less likely, after compilation errors, to blindly change parts of their code.

*CYNTHIA* can be downloaded from `http://ase.arc.nasa.gov/whittle/`

## References


1. A. Bundy, F. van Harmelen, C. Horn, and A. Smaill. The Oyster-Clam system. In M. E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 647–648. Springer-Verlag, 1990. Lecture Notes in Artificial Intelligence No. 449. Also available from Edinburgh as DAI Research Paper 507.
2. W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry; Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980.
3. David McAllester and Kostas Arkoudas. Walther recursion. In M. A. McRobbie and J. K. Slaney, editors, *13th International Conference on Automated Deduction (CADE13)*, pages 643–657. Springer Verlag LNAI 1104, July 1996.
4. R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.
5. J.N.D. Whittle. *The Use of Proofs-as-Programs to Build an Analogy-Based Functional Program Editor*. PhD thesis, Division of Informatics, University of Edinburgh, 1999.